Visual Programming as a Means of Communication in the Measurement System Development Process

Ed Baroth, Ph.D. & Chris Hartsough, M.S.
Measurement Technology Center
Jet Propulsion Laboratory
California Institute of Technology
E-mail: ebaroth@inst-sun1.jpl.nasa.gov

INTRODUCTION

This article reports direct experience with two commercial, widely used visual programming environments, National Instruments' LabVIEW and Hewlett Packard's VEE (Visual Engineering Environment). It compares visual programming with text-based programming in the environment of test and measurement (including simulation and data analysis). in this environment, we have found that visual programming currently provides productivity improvements of from four to ten times compared to conventional text-based programming [1,2]. The most dramatic gains in productivity are attributed to the communication among the customer, developer, and computer that are facilitated by the visual syntax of the tools.

The Measurement Technology Center (MTC) evaluates commercial data acquisition, analysis, display and control hardware and software products that are then made available to experimenters at the Jet Propulsion Laboratory. The MTC specifically configures and delivers turn-key measurement systems that include software, user interface, sensors (e.g., thermocouples, pressure transducers) and signal conditioning, plus data acquisition, analysis, display, simulation and control capabilities [3,4].

Visual programming tools are frequent>' used to simplify development (compared to text-based programming) of such systems, specifically LabVIEW and HP VEE. Employment of visual programming tools that control off-the-shelf interface cards has been the most important factor in reducing time and cost of configuring these systems. The MTC consistently achieves a reduction in software/system development time compared to text-based software tools tailored specifically to our environment [5,6]. Others in industry are reporting similar increases in productivity and reduction in software /system development time and cost [7,8].

Our use of visual programming provides an environment where programs (not simply user interfaces) are produced by creating and connecting icons, instead of traditional text-based programming. The icons represent functions (subroutines) and are connected by 'wires' that are paths which variables travel from one function to the next. Visual 'code' is actually the diagram of icons and wires rather than a text file of sequential instructions. Previous terms for this type of environment have included diagrammatic, iconic or graphical programming. The tools discussed here are based on the data flow diagram (DFD) paradigm [9].

The results of using visual programming include increased productivity and customer acceptance of both our products and processes. The key feature of both systems is that they implement a visual syntax. LabVIEW and VEE blend the visual syntax's of data flow diagrams and flow charting; they also include (or are strongly influenced by) Hierarchical Input Process Output (HIP()) charting and, for VEE at least, a bit of decision tables, Both systems support the use of text for labels, notes, and expressing mathematical formulae.

Both environments share salient characteristics, and the results from using both are comparable. The visually based syntax is the key factor in the acceptance of the tool by our customers. Our experience is that the development paradigm of a 'Requirements' definition followed by an implementation phase is obsolete in our test and measurement environment, The process now more closely represents rapid applications development (l{ AD) [1 O], and eliminates a separate implementation phase because, in general, when the requirements definition has been completed, so has the system. Traditionally, the Requirements definition is part of the communications chain that ultimately ends with the developer coding at the computer. Using these tools shortens the communications chain between customer, developer, and computer because coding usually is implemented interactively with the customer and developer together at the computer.

Our evidence shows that these tools facilitate communications because they provide a common expression that can be read by our customers, the developer, and the computer. There are different details of each syntax that facilitate this communication, but the details are unimportant: what is important is the transformation of requirements from a statement to a dynamic conversation that results in system components as a natural outcome of the process.

Our entry into visual programming environment came through vendors of test equipment. We began using these tools simply as a better way to meet our customers needs. Since the initial uses, we have become aware of more general uses of the paradigm and are now expanding our use of these tools and investigating the applicability of other visual and object oriented programming environments (e. g., Prograph). We recognize that these are not the only visual programming environments, but these are the two with which we have extensive experience to date [11]. This article will focus on LabVIEW and its applications, however, as VEE does not run on the Macintosh. More detail on Hewlett l'ackard's VEE is available from other sources [6,1 2,13].

National Instruments LabVIEW

LabVIEW is a graphics-based language environment for developing, debugging, and running programs. Initially designed to work with National Instruments' data acquisition and control boards that plug into a Macintosh, it has now been expanded to the PC, Sun and HP platforms. Because it is closer to a general purpose language, however, it can and has been used for many different types of applications, including simulations and pure data processing and analysis. The first version of LabVIEW appeared in 1986 and was interpreted and monochrome. The LabVIEW 2 compiler was released in 1990 and supports color. LabVIEW 3.0 supports

cross-development between the Mac, PC, HP and Sun and look and operate essentially the same way on all three platforms,

The LabVIEW program is used to create and run LabVIEW document files that are called Vi's (Virtual Instruments). The front panel of the VI appears in a window when opened and may contain an assortment of input and output objects such as knobs, dials, meters, charts, animated graphics and text boxes. Inputs are called controls and outputs are called indicators. The front panel may also contain passive graphics and text (Figure 1).

Associated with the VI is an icon that can be any small graphic image. 'Behind' the icon is a connector pane that can have an active region associated with each control and indicator on the front panel.

in addition to the front panel of each VI is a diagram that appears in another window when opened. The diagram contains an icon for each control and indicator. These icons are 'wired' together or to other built-in icons representing various functions and structures or to other VI files that have previously been 'collapsed' into their own icons and are referred to as subVI's (subroutines).

The process of developing a VI starts with creating a front panel with the required controls and indicators. If the intent of the VI is to function as a user interface, then emphasis may be placed on visual impact and usability. If the VI will be used primarily as a subVI, with parameters passed to and from it by other Vi's, then simple numeric controls and indicators may be used instead.

The programmer then typically works in the diagram window, developing the overall structure of the program using the built-in icons. Many of these icons are dynamic, in that they can be expanded to accommodate more inputs or outputs or re-sized to provide more area for other icons and wires (Figure 2). Other icons are added as needed and any that represent subVI's that haven't been developed yet can also be included as dummy functions that will automatically switch over to the actual subVI's as they become available. This feature can also be used to simulate hardware functions in the early stages of programming and then switched over to the actual hardware interfaces.

At any stage of the programming process, if the 'RUN' icon for the VI does not appear broken, the programmer can test the VI. LabVIEW will automatically compile and execute the VI. After each test, changes can be made on the front panel or the diagram or to any subVI's. Most LabVIEW programming is done with the mouse rather than the keyboard.

For more details on the LabVIEW environment, other sources are available [6,1 4,1 5].

EXAMPLE APPLICATION USING LABVIEW

Figures 1 and 2 are from an application using a Macintosh (Quadra 950) computer that is being used to simulate, test and display a telemetry stream. Currently, the MTC is supporting a software redesign of the computer system aboard the Galileo spacecraft [16]. To assure that every byte is correctly downloaded, the ground Test Bed setup of the computer subsystems (which mimics the computers

aboard the spacecraft) and the emulation hardware for the instruments is monitored. The performance of any new software is assessed by the LabVIEW code by checking the telemetry for accuracy.

The analyzer (Figure 1) is in current use monitoring telemetry from the Galileo spacecraft Test Bed. It provides easy visibility into the decommutation process modified by the Galileo programming support team. The time to write and modify the code using visual programming was significantly less than using text-based code.

The total number of control and indicators on the front panel is 475, which explains why it takes more than four minutes to compile (a rather long time by LabVIEW standards) after any changes. The main program has 100 subroutines and requires 7 Megabytes of disk space.

The program or diagram is shown in Figure 2. This is the actual program that LabVIEW runs. It contains several structures that enable looping, sequencing, and selecting among several cases, as well as icons representing the controls and indicators on the front panel, and subroutines. It is explained in more detail in reference 16.

This task demonstrated a dramatic increase in productivity and reduction in schedule as well as verifying the approach of using visual programming for realistic and relatively complicated applications. The end-user believes no other programming approach could produce this level of output, due in part to much of the programming being done with the customer and programmer together at the computer.

Other advantages demonstrated were in the areas of prototyping and verification. Different approaches were demonstrated and evaluated quickly using a visual programming language. Verification can be demonstrated using the graphical user interface features available in a visual programming language easier than using conventional text-based code.

VISUAL PROGRAMMING AND COMMUNICATION

The advantages/disadvantages of any programming environment are dependent on the context in which they are being used. Our specific environment is the production of measurement systems, usually under schedule pressure, in four years, the MTC has created over sixty applications, from the intended uses of data acquisition and control to areas not originally intended including simulation, analysis, telemetry, training, and modeling. We have found productivity increases (compared to text-based tools) in all applications, domain specific or not.

It is significant to note that the productivity gains are not the result of a basic paradigm shift. As stated, both LabVIEW and VEE directly implement hybrids of the data flow diagram paradigm. Both tools, in effect, collapse the phase called 'coding' because the diagram executes. Programming, of course, is still taking place, but there is no programming activity as it is integrated with the requirements discovery and systems design process. Keeping a well-known paradigm has both positive and negative effects. in the plus column is ease of learning, ease of communication with

the user, speed, and adaptability. In the minus column are mostly implementation effects, excluding the major limitation: the underlying paradigm.

A limitation of the discussed visual programming tools is they are based on the data flow diagram paradigm. For problems that won't yield to a data flow diagram analysis, these tools are not particularly useful. Neither tool produces a conventions] text-based programming language representation of the model. For many programmers, this is perceived as a major disadvantage and in some cases precludes acceptance. The authors have not found this to be a problem in our use of the tools.

There have been few studies comparing visual with other types of programming, and those that do exist have focused on aspects that do not seem to correspond with our use of visual programming in the real world. The study by Green et al. [17] compared readability of textual and graphical programming (LabVIEW). Their clear overall result was that graphical programs took longer to understand than textual ones. The study by Moher et al. [18] essentially duplicated the study by Green et al. but compared petri-net representations with textual program representations. They duplicated some of the earlier results, but did find areas where the petri-net representation was more well suited, albeit with reservations.

Both studies focused on experienced users of visual or textual code. In neither study was the time to create or modify the programs discussed. It is in these areas, that of user (not programmer) experience and time to create and modify programs, that we find advantages in visual over textual programming in our real world. in addition, both studies used only static visual representations, whereas in real world systems, customers and developers get to interact with the program while trying to understand it.

Our customers are mostly engineers and scientists with limited programming experience with either visual or text-based code. Most, if not all, understand data flow diagrams, so the question becomes one of which representation is easier to understand with little or no prior experience. We have consistently found users with little or no experience in LabVIEW or VEE could 'understand' at least the process, if not the details, of the program. in fact, we usually program together with the customer at the terminal, and they follow the data flow diagrams enough to make suggestions or corrections in the flow of the code. It is difficult to imagine a similar situation using text-based code, where someone with little or no understanding of 'C' could correct a programmer's syntax or flow. Actually, it is difficult to imagine anyone 'watching' someone else program using text-based code at all.

The study by Pandey and Burnett [19] did compare time, ease and errors in constructing code using visual and text-based languages. The programs chosen were on the level of 'homework' type tasks, certainly not real world problems, but even at that level they did find evidence that matrix and vector manipulation programs were more easily constructed and had fewer errors using visual programming.

Using visual programming at this last stage of the coding process, however, removes much of the advantages we've seen. Once specifications are determined, it

simply becomes a race to see who can type faster or who has access to more or better libraries of code or icons. The real benefit we find in using visual programming is the flexibility in the design process, before requirements have been determined. The user-programmer-computer communication is substantially improved because of the speed at which modifications can be made. None of the existing studies have dealt with the ability of visual or text-based programming to solve real world problems, i.e., to determine specifications, and operate and modify code and user interfaces, as well as train inexperienced users to operate and modify systems.

The most important advantage the MTC has found in using visual programming is the support for communications among the customer, developer, and hardware that visual programming enables. This ease of communication provides the ability to go from conception to simulation of components, sub-systems and systems, to testing of actual hardware and control functions using a single software environment (on multiple platforms). Modules or icons that represent simulations of instruments, processes or algorithms can be easily replaced with the actual instruments or components when they become available.

These icons include third-party software products that take advantage' of the existing visual syntax of the tools, as opposed to learning an additional 'language' for each application. We expect to see more (some exist already) data analysis, data visualization, data display or database icons that allow the user to access their programs directly using LabVIEW or VEE. This will enhance the communication process as it brings data acquisition through data display and presentation together using a single visual programming environment. Whenever a user has to switch programming environments to change applications, the speed of communications is reduced.

The fundamental limits of these tools are scaling and maintenance. The MTC has produced real systems of moderate scope. We have not hit the scaling limit yet, but it is clearly present. Our systems have short to intermediate lifespans, a few weeks to a few years. Our customers tend to maintain the systems we develop for them, but have not attempted major revisions without support of the original author. If these systems had to be maintained over ten years, we're not certain that our current implementation techniques would be adequate.

These tools arc best used on problems that are functionally intensive, not data intensive. These systems excel in transformation and display of volatile data sources, as opposed to the maintenance of large data repositories. Currently they are not appropriate for image data, although they could be extended into that arena comfortably. Beyond this, we are reluctant to bias a reader away from any area: we've been successful in areas that were purported to be inappropriate too many times.

CONCLUSIONS

- **Both visual programming tools we have used provide comparable productivity improvement**

The DFD **environment** (not simply one program from one developer) has shown rea capability of reducing software development time in areas not domain

specific. This productivity improvement is due primarily to the improved communication between the customer, developer, and computer that the visual syntax provides, If the visualization component of a programming system does not support the customer, developer, and computer communication, the productivity improvements associated with LabVIEW and VEE will not be present,

- **Existing system development methodologies are inadequate in this new environment.**

Because existing system development methodologies presume the existence of one or more coding phases, and that these phases are conducted outside the presence of the customer, they do not address the work environment that we find ourselves in. The lack of viable methodology is not a simple issue. If you simply compare coding time between a visual and text language you miss the point. Using these systems essentially blurs the requirements, design, and coding phases into a single activity. in many cases the MTC has found that it is faster to build the system using informal specifications than to write a formal requirements document to then build the system.

The environment of visual programming has changed the communication between developer and customer. Instead of communicating in writing or meetings, the definition of requirements takes place using visual programming while the 'code' is being diagramed. Development becomes a joint effort between developer and customer. in these working sessions, it is often the developer that waits while the customer considers what is wanted or what next needs to happen. Some development still occurs without the customer's participation, e.g., questions concerning the operating system interfaces or when no immediate feedback from the customer is required.

What to do with these patterns in the context of a conventional development model is unclear. This is a serious issue. There are almost no predictors of job resource requirements: by the time the traditional measures are available, the job is nearly done. It is difficult to manage these projects because there is no realistic model against which to measure progress. So far, the only measures we use consistently are measures of system behavior.

We are doing tasks that are not small, but not very large either. When we 'scale up' for larger projects, issues of predictable methods will become more serious.

- **The *visual* aspect of these tools is not an add-on but integral to the underlying method of expression.**

Both LabVIEW and VEE are tools that automate a graphic syntax already in common use. Within both are features that have been adapted from text paradigms. Where the text form is imported directly, e.g., FORTRAN or C equation expressions, it works well. When a basic text construct such as data structure has graphics components appended to a well-understood text syntax, the whole thing falls a bit flat. Some attempts to put object oriented features in a graphical language have had

some of the same problems, i.e., graphics were simply added and not part of an underling graphical syntax. This is not to say that the graphics don't help, they do, it is just that the results are not as dramatic as automating graphic syntax directly.

- **Without the *visualization* component of these tools in viewing program execution, the tools would be of limited or no value.**

Visualization **in this context is the ability** to **graphically communicate the state** of execution of a system to the customer. This capability to see what the 'code' is doing directly is of inestimable value. The graphics description of the system without the animation would be not much more than a CASE tool with a code generator; with the animation, the boundaries between requirements, design, development, and test appear to collapse. Seamless movement from one activity focus to another makes the development different in kind, not degree. This is because we can sustain the communication among the customer, developer, and computer. If there were substantial time lags in changing tools, (e.g., conventional debuggers) the conversational environment would break down.

- **Failure to incorporate standard hardware drawing control capabilities places a burden on the memory (mental and paper) of the developers / maintainers of very large systems.**

Managing large sets of drawings using parts lists and reference designators is not new. Configuration management support in visual languages is not yet present. The single largest problem we face in scaling up the use of these tools into larger systems devolves to configuration management, Presently, there is no clear answer to this problem.

- **The tools facilitate, they don't provide the solutions.**

in the hands of an expert (in both development and problem domains), these tools provide tremendous leverage on time and efficiency. in the hands of a novice (in either area), you still have a novice. Part of being an expert is knowing when to switch tools.

ACKNOWLEDGMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The authors wish to acknowledge the contribution of George Wells toward the writing of this article.

REFERENCES

[1] G. Wells and E. C. Baroth, "Telemetry Monitoring and Display using LabVIEW," *Proceedings of National Instruments User Symposium*, Austin, Texas, March 28-30, 1993.

[2] D. Breeman, "Jet Propulsion Lab Aids in Space Craft Project," *Scientific Computing and Automation*, November, 1993, pp. 26-28.

[3] E. C. Baroth, D. J. Clark, and R. W. Losey, "Acquisition, Analysis, Control, and Visualization of Data Using Personal Computers and a Graphical-Based Programming Language, " *Conference Proceedings of American Society of Engineering Educators (A SEE)*, Toledo, Ohio, June 21-25, 1992, pp. 1447-1453.

[4] E. C. Baroth, D. J. Clark, and R. W. Losey, "An Adaptive Structure Data Acquisition System using a Graphical-Based Programming Language," *Fourth AIAA/Air Force/NA SA/OAI Symposium on Multidisciplinary Analysis and Opt imization*, Cleveland, Ohio, September 21-23, 1992.

[5] E. C. Baroth, C. Hartsough, L. Johnsen, J. McGregor, M. Powell-Meeks, A. Walsh, G. Wells, S. Chazanoff, and T. Brunzie, "A Survey of Data Acquisition and Analysis Software Tools, Part 1," *Evaluation Engineering Magazine*, October, 1993, pp. 54-66.

[6] E. C. Baroth, C. Hartsough, L. Johnsen, J. McGregor, M. Powell-Meeks, A. Walsh, G. Wells, S. Chazanoff, and T. Brunzie, "A Survey of Data Acquisition and Analysis Software Tools, Part 2," *Evaluation Engineering Magazine*, November, 1993, pp. 128-140.

[7] *Proceedings of National instruments User Symposium*, Austin, Texas, March 28-30, 1993.

[8] *Proceedings of National instruments European User Symposium*,, Munich, Germany, November 9-11, 1994.

[9] J. Kodosky, J. MacCrisken, and G. Rymar, "Visual Programming Using Structured Data Flow," *Proceedings of the 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 8-11, 1991, pp. 34-39.

[10] E. Yourdon, *Decline and Fall of the American Programmer*, Yourdon Press, Prentice Hall inc., Englewood Cliffs, 1992.

[11] Baroth, E. C. and Hartsough, C., "Experience Report: Visual Programming in the Real World," *Visual Object Oriented Programming*, edited by M. M. Burnett, A. Goldberg & T. G. Lewis, Manning Publications, Prentice Hall, 1995, PP" 21-42"

[12] R. Helsel, *Cutting Your Test Development Time With HP VEE*, Prentice Hall inc., Englewood Cliffs, 1994.

[13] "VEE Visual Engineering Environment," Hewlett Packard Technical Data, 5091 -1142EN, 1991.

[14] Johnson, G. W., *LabVIEW Graphical Programming*, McGraw-Hill, New York, 1994

[15] *National Instruments Catalog, 1995, pp.* 17-112.

[16] G. Wells and E.C. Baroth, "Using Visual Programming to Simulate, Test, and Display a Telemetry Stream," MacSciTech's SEAM '95 Conference, San Francisco, California, January 8-9, 1995.

[17] T. R.G. Green, M. Petre, and R. K. E. Bellamy, "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture," *Fourth* Workshop *on Empirical Studies of Programmers,* New Brunswick, New Jersey, December 7-9, 1991, pp. 121-146.

[18] T. G. Moher, D. C. Mak, B. Blumenthal, and L. M. Leventhal, "Comparing the Comprehensibility of Textual and Graphical 1 'rograms: The Case of Petri Nets," *Fifth Workshop on Empirical Studies of Programmers,* Palo Alto, California, December, 3993.

[19] R. K. Pandey and M. M. Burnett, "Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Imperical Study," Oregon State University, Department of Computer Science, 93-60-08.

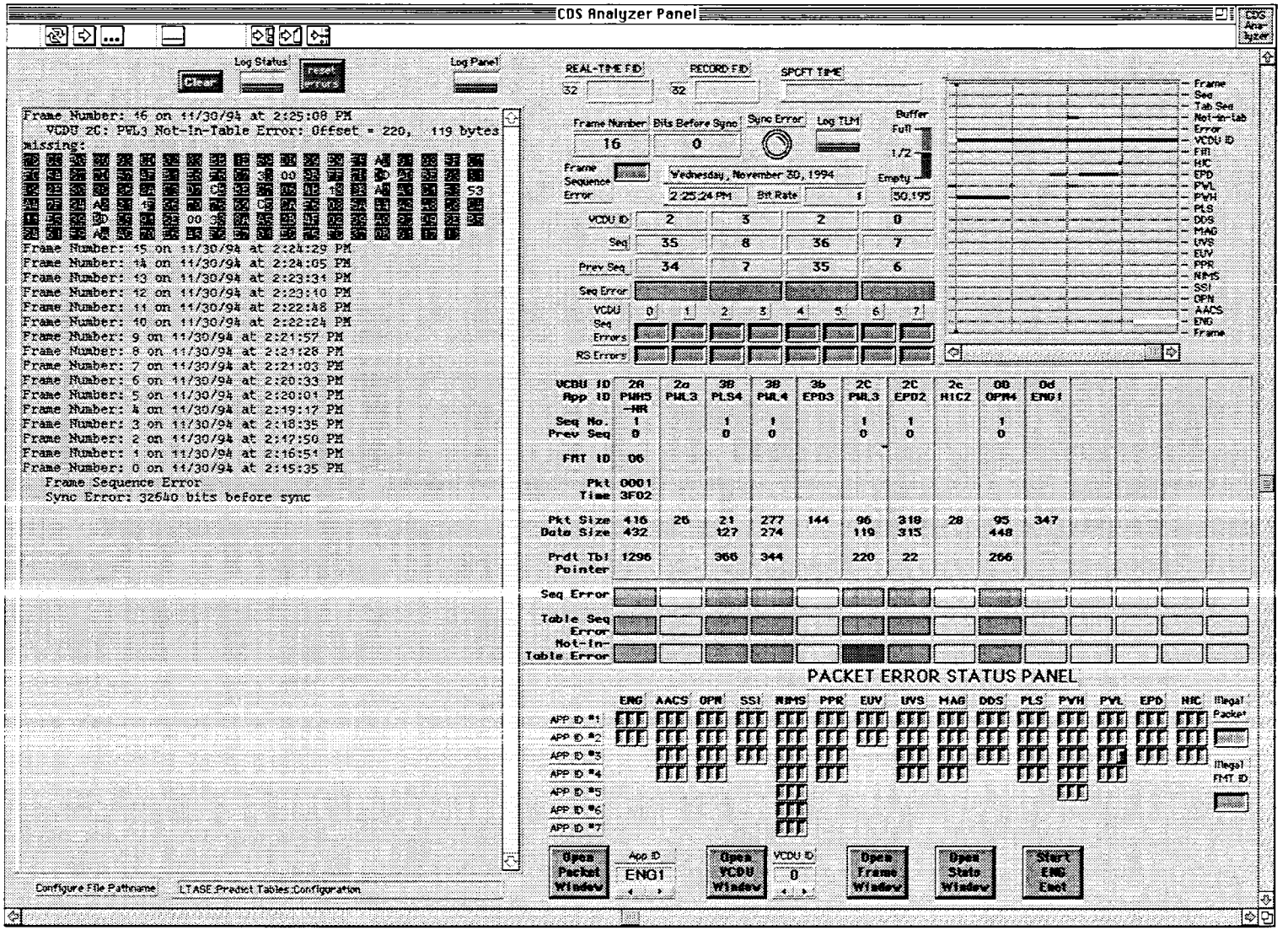

Figure 1. LabVIEW User Interface of Telemetry Analyzer

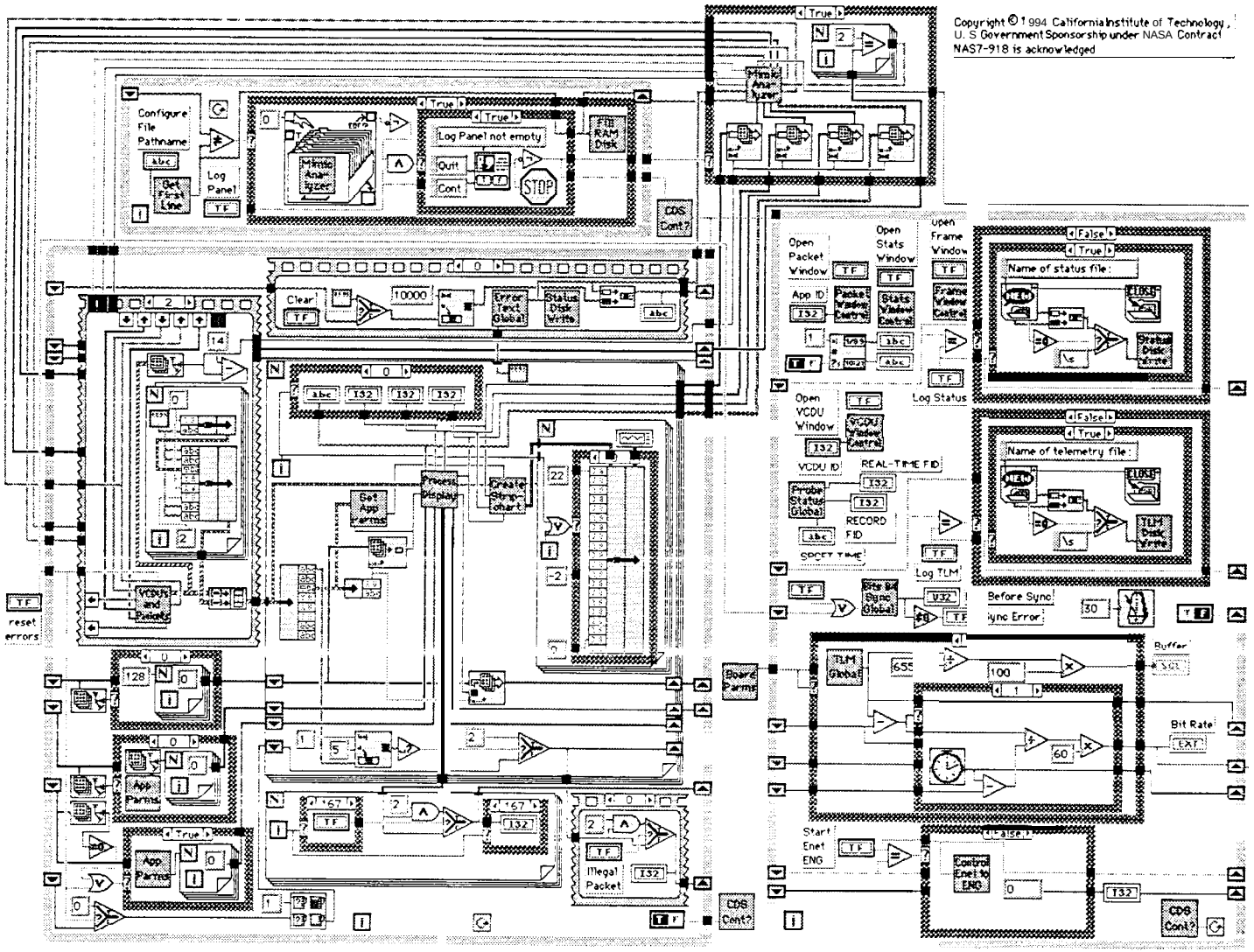

Figure 2 Analyzer program (LabVIEW diagram)